5. Analyse d'algorithmes

LA COMPLEXITÉ D'UN ALGORITHME mesure sa performance en fonction de la taille du problème il est destiné à résoudre. En particulier, on caractérise les ressources (temps, mémoire) nécessaires pour l'exécution.

- ⋆ **usage de mémoire** ou «complexité d'espace» (*space complexity*) : c'est le mémoire de travail nécessaire à part de stocker l'entrée même.
- ⋆ **temps de calcul** ou «complexité de temps» (*time complexity*) : c'est le temps d'exécution dans un modèle formel de calcul.

La théorie de complexité[1] est une des branches importantes de l'informatique : elle étudie les mathématiques de la difficulté intrinsèque de problèmes. Dans ce cours, on préfère des caractérisations de temps ou de mémoire avec utilité pratique. (Par exemple, on veut prédire/expliquer ce qui se passe si on fait rouler l'algorithme sur $n = 10^6$ éléments.)

[1] W(fr):Théorie de complexité

5.1 Analyse de temps de calcul

Afin de développer une caractérisation d'utilité pratique, on procède comme suit.

1. Développer un modèle de l'entrée et définir la notion de la «taille» de l'entrée. Le modèle doit permettre la génération de données à l'entrée pour mesurer la performance d'une implantation sur l'ordinateur.
2. Identifier la partie du code le plus fréquemment exécutée (qui domine la croissance du temps de calcul). Pour un algorithme itératif, cette partie est à l'intérieur de la boucle le plus profondement imbriquée.
3. Définir un modèle de coût pour le temps de calcul. En particulier, on veut écrire le temps de calcul avec le coût d'opérations typiques dans le contexte du problème. Exemples d'opération typique : comparaison de deux éléments lors du tri d'un fichier, accès à une cellule dans un tableau, ou une opération arithmétique (algorithme d'Euclid, exponentiation).
4. Déterminer la fréquence d'exécuter les opérations typiques en fonction de la taille de l'entrée.

Exemple 5.1. On prend le fameux exemple de chercher le minimum dans un tableau par itération sur les éléments. En suivant la recette : 1. taille = longueur de tableau n, 2. le cœur du code est la comparaison entre $x[i]$ et min à l'intérieur de la boucle, 3. coût $C(n)$ = nombre de comparaisons $x[i] < \mathsf{min}$; 4. $C(n) = n - 1$. ♠

```
1 initialiser min ← x[0]
2 for i ← 1,...,n − 1 do
3 |   if x[i] < min then min ← x[i]
4 end
5 return min
```

FIG. 1: Algorithme itératif pour chercher le minimum.

Meilleur, pire, ou moyen. Typiquement, il y a beaucoup d'entrées possibles avec la même taille n, et le temps de calcul peut dépendre de l'entrée actuelle et non pas seulement de sa taille. Dans une telle situation, la caractérisation en 4 peut montrer le temps maximal (**pire cas**), le temps minimal (**meilleur cas**) ou le **moyen cas**. Ce dernier assume forcément une distribution.

Exemple 5.2. Soit $D(n)$ le nombre de déplacements dans tri par insertion sur un tableau de taille n. Dans le meilleur cas (tableau trié), $\min D(n) = 0$. Au pire (trié dans le sens inverse), $\max D(n) = \sum_{i=0}^{n-1} i \sim n^2/$. Assumant une permutation uniforme, le nombre moyen est $\bar{D}(n) = \sum_i \sum_{j=0}^{i} j/i \sim n^2/4$, malheureusement quadratique. ♠

Expériences

Une bonne caractérisation du temps de calcul nous permet de le considérer comme un *hypothèse scientifique* testable sur le comportement de l'algorithme.

Après qu'on a implanté l'algorithme, on peut expériencer avec des entrées différentes, en mesurant le temps, ou comptant les opérations typiques sur des entrées différentes.

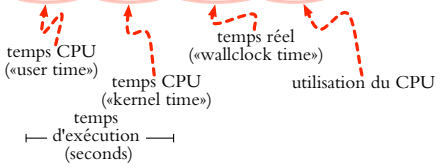
Mesurer le temps de calcul. Le temps d'exécution peut être mesuré :

- ⋆ dans le code (Java) :

```
...
long T0 = System.currentTimeMillis(); // temps de début
...
long dT = System.currentTimeMillis()-T0; // temps (ms) dépassé
```

- ⋆ dans le shell (Linux/Unix) avec la commande `time` :

```
% time java -cp Monjar.jar mabelle.Application
0.283u 0.026s 0:00.35 85.7%  0+0k 0+53io 0pf+0w
```



- ⋆ ou bien dans un environnement de développement de logiciel (Netbeans, Eclipse) qui supporte *profiling*.

Compter les opérations. Afin de compter les opérations :

- ⋆ modifier le code et incorporer des variables de compteur

```
private static final boolean COUNT=true;
private static int op_count=0;
private static void insert(double[] T, int n, double x){
    // T[0]<=T[1]<=...<=T[n-1] déjà trié
    int i=n;                              // indice d'insertion de x
    while(i>0 && T[i-1]>x) {              // décalage
      T[i]=T[i-1]; if (COUNT) $op_count++;
      i--;
   }
   T[i]=x;
}
public static void insertionSort(double[] T){
   for (int n=0; n<T.length; n++) insert(T,n,T[n]);
   if (COUNT) System.out.println("#COUNT\t"+T.length+"\t"+op_count);
}
```

- ⋆ profiler avec environnement de développement de logiciel

Conception d'expériences.

Dans une étude empirique, on répète les expériences avec des entrées différentes (simulées ou benchmark). (1) On répète l'expérience pour entrées différentes de même taille. La moyenne montre le comportement typique, et les répétitions permettent de comprendre la dispersion statistique du temps de calcul. (2) On répète l'expérience pour entrées de taille différente. Il est particulièrement utile de considérer des tailles multipliées par la même facteur (2 ou 10). Si on a l'hypothèse que le temps de calcul est $T(n) \sim a \cdot n^b$, avec des constantes quelconques $a, b > 0$, on a

$$\frac{T(2n)}{T(n)} \sim \frac{a(2n)^b}{an^b} = 2^b. \tag{5.1}$$

En fait, Equation (5.1) permet de déduire b par régression linéaire même si on ne le sait pas. On a $\log T(n) \sim a' + b \log n$, et donc on peut déterminer b par la pente dans un répère log-log (Fig. 2). Les mesures servent aussi à prédire (par extrapolation) le temps de calcul de l'implantation examinée pour des entrées de grande taille.

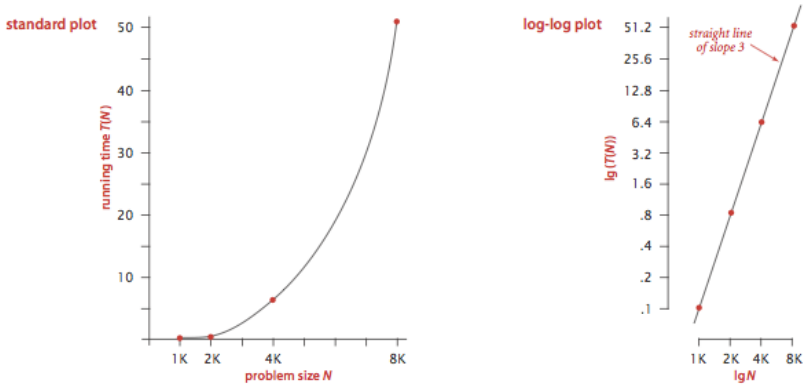


FIG. 2: Temps de calcul mesuré et tracé sur un graphe. L'échelle log-log permet de visualiser des dépendences polynomiales. Ici, c'est une fonction $T(N) \sim aN^3$.

Diviser pour régner

Le temps de calcul d'un algorithme récursif s'écrit typiquement par récursion.

La technique de «**diviser pour régner**» (*divide-and-conquer*) exploîte la nature récursive de solutions optimales à une classe de problèmes. La démarche générale est de (1) couper le problème dans des sous-problèmes similaires, (2) chercher la solution optimale aux sous-problèmes par récursion, (3) combiner les résultats. Quand un problème de taille n est coupé en m sous-problèmes de taille n_i: $i = 1, \ldots, m$, le temps de calcul est déterminé par la récurrence $T(n) = \tau_{\text{partition}}(n) + \sum_{i=1,\ldots,m} T(n_i) + \tau_{\text{combiner}}(n)$, où $\tau_{\text{couper}}(n)$ et $\tau_{\text{combiner}}(n)$ sont les temps pour couper et combiner.

```
1 MinRec(x[0..n−1], g, d)
2 // minimum parmi x[g..d−1]
3 if d − g = 0 then return ∞
4 if d − g = 1 then return x[g]
5 mid ← ⌊(d + g)/2⌋
6 m1 ← MinRec(x, g, mid)
7 m2 ← MinRec(x, mid, d)
8 return min{m1, m2}
```

FIG. 3: Algorithme récursif pour chercher le minimum. Premier appel : MinRec$(x, 0, n)$.

Exemple 5.3. On prend l'exemple de chercher le minimum avec une approche diviser-pour-régner. Figure 3 montre cette solution : couper le tableau en deux et chercher les minimums dans les deux sous-tableaux. On mesure le calcul par $C(\ell)$, le nombre de comparaisons entre éléments (en Ligne 8) quand $d - g = \ell$. Par l'inspection du code, on a

$$C(\ell) = \begin{cases} 0 & \{\ell = 0, 1\} \\ C(\lfloor \ell/2 \rfloor) + C(\lceil \ell/2 \rceil) + 1. & \{\ell > 1\} \end{cases} \tag{5.2}$$

La solution est $C(\ell) = \ell - 1$ pour tout $\ell > 0$ ce qu'on peut démontrer par induction. **Cas de base :** $C(1) = 1 - 1 = 0$. **Hypothèse d'induction :** $C(k) = k - 1$ pour tout $0 < k < \ell$ à $\ell > 1$. **Cas récursif :**

$$C(n) = \underbrace{\lfloor \ell/2 \rfloor - 1}_{\text{H.I.}} + \underbrace{\lceil \ell/2 \rceil - 1}_{\text{H.I.}} + 1 = \ell - 1.$$

Conclusion. $C(\ell) = \ell - 1$ pour tout $\ell > 0$. ♠

Étude empirique d'une récurrence. D'autres récurrences peuvent être plus difficile à manipuler. On prend l'exemple de la récurrence suivante qui caractérise le nombre de comparaisons dans le tri par fusion :

$$C(n) = \begin{cases} 0 & \{n = 0, 1\} \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + (n - 1) & \{n > 1\} \end{cases} \tag{5.3}$$

Cette récurrence n'a pas une solution explicite aussi ordinaire que (5.2). Mais on est des informaticiens, on peut écrire un petit code pour calculer les valeurs exactes à $n = 0, ..., N$, un seuil N adéquat. Ensuite, on peut tenter de trouver une fonction simple $f(n)$ qui capture la croissance de C. Figure 4 montre $C(n)$ sur répère log-log : aucune approximation linéaire ne reproduit pas la courbe observée. Par contre, on peut énoncer l'hypothèse que $C(n) \sim n \lg n$ à la base du graphe (rappel : $\lg = \log_2$). On peut inférer une approximation même plus précise en examinant la différence $C(n) - n \lg n$: les graphe suggèrent que $C(n) \sim n \lg n - n + \delta(n)$, où $\delta(n)$ est une fonction périodique qui ressemble une série de bosses, avec les maximums locaux sur une pente linéaire.

La solution exacte est en fait bien connue : $C(n) = n \lg n - nA(\{\lg n\}) + 1$ où $\{\lg n\} = \lg n - \lfloor \lg n \rfloor$ et $A(u) = u + 2^{1-u} - 1 \in [0.9139 \cdots, 1]$

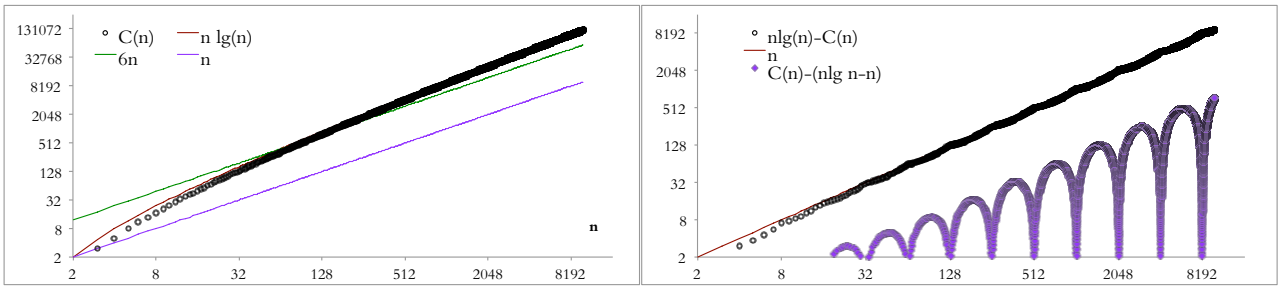


FIG. 4: Number of comparisons $C(n)$ in mergesort.

5.2 *Notation asymptotique*

Une définiton récursive comme (5.3) est exacte, mais elle n'est pas utile pour caractériser comment le temps de calcul croît avec la taille du problème. En algorithmique, on préfère des caractérisations asymptotiqes, en termes de fonctions simples : $C(n) \sim n \lg n$. L'**échelle standard** inclut les fonctions

- ⋆ pouvoirs de n : $g(n) = n^a$ incluant $a = 0$,
- ⋆ logarithmes et logarithmes itérés : $g(n) = \log n, \log\log n, \log\log\log n, \ldots$,
- ⋆ exponentiales de n : $g(n) = a^n$ avec une constante a,

et leurs produits.

Le plus souvent, on travaille avec des approximations asymptotiques de forme $T(n) \sim c \cdot f(n)$ où $T(n)$ est la quantité (temps de calcul) caractérisée, c est une constante et $f(n)$ est une fonction sur l'échelle standard : $f(n) = n^a \log^b(n)$ qui s'appelle l'**ordre de croissance** de $T(n)$. La base du logarithme est sans importance dans l'ordre de croissance car la constante peut absorber le changement de base : $\log_a x = \log_b x / \log_b a$.

ordre	$f(n)$
constante	1
logarithmique	$\log n$
linéaire	n
linéarithmique	$n \log n$
quadratique	n^2
cubique	n^3
exponentiel	A^n

FIG. 5: Ordres typiques dans l'analyse de structures de données

Notation de Landau

L'ordre de croissance d'une fonction est exprimée mathématiquement par une notation spécifique de grand-O, Θ, Ω et petit-o[2]. La définition utilise «presque» dans un sens précis : «**presque tout**» veut dire qu'il y a juste un nombre fini (même aucune) d'exceptions. Ceci permet, entre autres, d'ignorer

[2] W(fr):Grand-o, petit-o

les aléas de la fonction aux valeurs très petites.

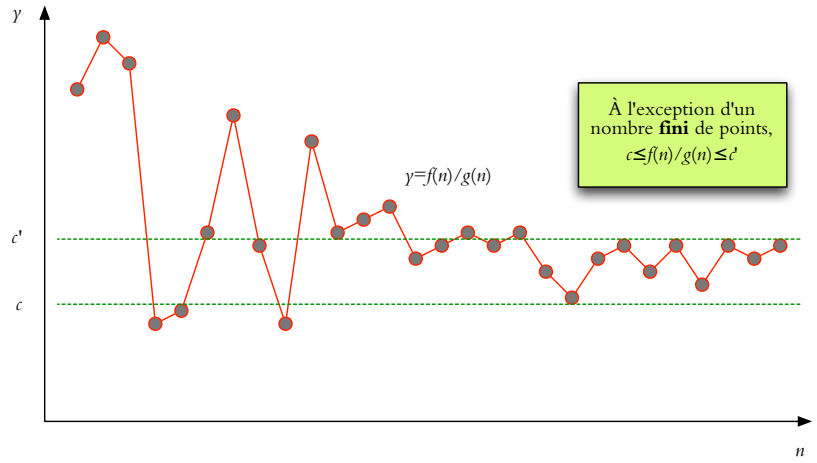


FIG. 6: Illustration de «presque tout» pour $f = \Theta(g)$.

Définition 5.1. *Soit f et g deux fonctions sur les nombres entiers telles que $f(n), g(n) > 0$ pour presque tout n.*

[grand O] *$f = O(g)$ si et seulement si [ssi] $\exists c > 0$: tel que $\frac{f(n)}{g(n)} \leq c$ pour presque tout n.*

[grand Omega] *$f = \Omega(g)$ ssi ou $\exists c > 0$: tel que $\frac{f(n)}{g(n)} \geq c$ pour presque tout n. (Et donc $g = O(f)$.)*

[Theta] *$f = \Theta(g)$ ssi $f = O(g)$ et $g = O(f)$, ou $\exists c, c' > 0$ tels que $c \leq f(n)/g(n) \leq c'$ pour presque tout n. Voir Fig. 6*

[petit o] *$f = o(g)$ ssi $\lim_{n\to\infty} f(n)/g(n) = 0$, ou $\forall c > 0$, $\frac{f(n)}{g(n)} \leq c$ pour presque tout n*

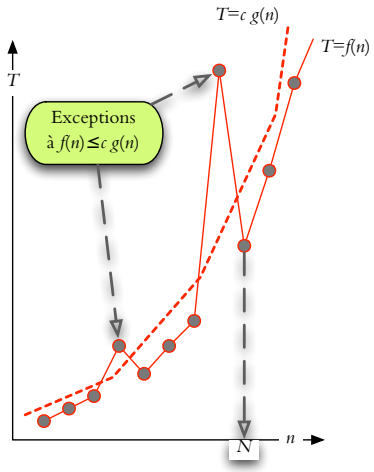


FIG. 7: Définition équivalente pour grand-O : $f = O(g)$ si et seulement s'il existe $c > 0$ et $N \geq 0$ tels que $f(n) \leq c \cdot g(n)$ pour tout $n \geq N$.

Expansion asymptotique

L'expansion asymptotique d'une fonction $f(n)$ exprime la convergence vers une fonction construite sur une échelle fixe de croissances.

Définition 5.2. *Une série de fonctions $g_0(n), g_1(n), \ldots$ avec $g_{k+1} = o(g_k)$ s'appelle une **échelle asymptotique**. La série*

$$f(n) \sim c_0 g_0(n) + c_1 g_1(n) + c_2 g_2(n) + \cdots$$

*s'appelle l'**expansion asymptotique** de f. L'expansion représente les formules raccourcies*

$$\begin{aligned} f &= O(g_0) \\ f &= c_0 g_0 + O(g_1) \\ f &= c_0 g_0 + c_1 g_1 + O(g_2) \\ &\ldots \end{aligned}$$

On peut juste garder autant de termes de l'expansion pour arriver à une précision acceptable. Pour nos buts, 1–3 termes suffisent en général. Avec la notation grand-O on peut spécifier la grandeur des termes ignorés, tandis que la notation tilde les cache complètement. Notez que la forme la plus simple $f = O(g)$ ne correspond pas à un hypothèse testable : grand-O est une borne supérieure avec des constantes cachées.

Quelques fonctions notables

Logarithmes.

$$\lg x = \log_2 x \qquad \text{et} \qquad \ln x = \log_e x \quad \{x > 0\}$$
$$\log(xy) = \log x + \log y; \log(x^a) = a \log x$$
$$2^{\lg n} = n; n^n = 2^{n \lg n}; \quad \log_a n = \frac{\lg n}{\lg a} = \Theta(\lg n) \quad ; a^{\lg n} = n^{\lg a}$$

Dans la notation asymptotique, on ne montre pas la base du logarithme parce qu'avec une base différente, on change seulement la facteur constante ($\log_a f = c \cdot \log_b f$ avec $c = \log_a b = 1/\log_b a$) : au lieu de $O(\log_{10} n)$ ou $O(\ln n)$, on écrit simplement $O(\log n)$.

Nombre harmonique. Le nombre harmonique[3] est la somme des inverses des n premiers entiers

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n} = \ln n + \gamma + O(1/n) \sim \ln n = \Theta(\log n)$$

où $\gamma = \lim_{n\to\infty}(H_n - \ln n) = 0.5772\cdots$ est la constante d'Euler.

[3] W(fr):Nombre harmonique

FIG. 8: Leonhard Euler (1707–1783)

Nombres Fibonacci. $F(n) = \frac{\phi^n}{\sqrt{5}} + O(\phi^{-n}) \sim \frac{\phi^n}{\sqrt{5}} = \Theta(\phi^n)$ avec $\phi = (1+\sqrt{5})/2 = 1.6\cdots$.

Factorielle. $n! = 1 \cdot 2 \cdots n$. Approximation de Stirling : $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n = \Theta\left(n^{n+1/2}e^{-n}\right)$. Même si par intution n^n semble dominer l'expression, on doit retenir tout exposant de n dans un terme quand on simplifie vers Θ. Par contre, $\sum_{k=1}^{n} \ln k = \ln(n!) = \big(1 + o(1)\big) n \ln n$, donc $\log(n!) = \Theta(n \log n)$.

Déterminer l'ordre de croissance

Règles d'arithmétique. Les équations suivantes sont valides avec O, Θ, Ω, et o.

$$c \cdot f = O(f) \tag{5.4a}$$
$$\underbrace{O(f) + O(g)}_{\text{pour tout } h = O(f) \text{ et } h' = O(g)} = \underbrace{O(f+g)}_{\text{il existe } h'' = O(f+g) \text{ t.q. } h + h' = h''} \tag{5.4b}$$
$$O(f) \cdot O(g) = O(f \cdot g) \tag{5.4c}$$

Algorithme sans récurrence. On peut exploiter les règles d'arithmétique directement dans l'analyse du pseudocode. Pour établir le temps de calcul f, on examine le code et additionne le coût de chaque instruction. On sépare f par ses termes (règle (5.4b)), et on ignore les facteurs constantes (règle (5.4a)).

Exemple 5.4. On prend l'exemple de rechercher le maximum dans un tableau.

```
MAX-ITER(x[0..n − 1])
M1  initialiser max ← −∞                          O(1)
M2  for i ← 0, . . . , n − 1 do                   O(1)
M3      if x[i] > max then max ← x[i]             O(1)   × n
M4  return max                                    O(1)
```

Le temps de calcul pour un tableau de taille n est $T(n) = O(1) + O(1) + n \cdot O(1) + O(1) = O(n)$. ♠

Exemple 5.5. L'algorithme d'Euclide est important dans des applications cryptographiques, où on calcule avec des entiers de grande taille (par exemple, 2048 bits). La taille de l'entrée est donc mesurée dans le nombre de bits pour représenter les paramètres : $\lg a + \lg b$.

```
GCD(a, b) // {b ≤ a}
E1  while (b ≠ 0)                                 O(1)
E2      c ← a mod b                               m(a,b)   × O(log b)
E3      a ← b; b ← c                              O(1)
E4  return a                                      O(1)
```

Il faut considérer le coût de division entière (mod) : Ligne E2 prend $m(a,b) = O\big((\log a)(\log b)\big)$ temps. Théorème 1.4 montre que le nombre d'itérations est borné par $O(\log b)$ (on cherche le nombre Fibonacci $F(k) \leq b < F(k+1)$; donc $k = \log_\phi b + O(1) = O(\log b)$). Par conséquence, l'algorithme d'Euclide s'exécute en $T(a,b) = O(1) + O(\log b) \times \Big(O(\log a \log b) + O(1)\Big) = O(\log^2 b \log a)$ temps, donc en temps *polynomial* dans la taille de l'entrée, même pour des entiers très grands. ♠

Substitution de variables.

Exemple 5.6. On prend l'exemple du calcul de puissances. On veut calculer x^n où $n \geq 0$ est un entier non-négatif et $x \in \mathbb{R}$. La clé est la récurrence

$$x^n = \begin{cases} 1 & \{n = 0\} \\ x^{n/2} \cdot x^{n/2} & \{n > 0, n \text{ est pair}\} \\ x \cdot x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} & \{n > 0, n \text{ est impair}\} \end{cases}$$

On analyse l'algorithme (Fig. 9) en calculant le nombre de récursions $R(n)$. On a $R(n) = R(\lfloor n/2 \rfloor) + 1$ pour $n > 0$. On trouve la solution par **substitution de variables** : on regarde le nombre de bits dans la représentation binaire de n. Si on dénote ce nombre par b, on a la récurrence $R'(b) = R'(b-1) + 1$ qui donne $R'(n) = n - 1$ en considerant $R'(1) = 0$. Or, $b = \lceil \lg(n+1) \rceil$, alors $R(n) = \lceil \lg(n+1) \rceil - 1 = O(\log n)$, L'algorithme donc prend un temps logarithmique dans n et dans sa taille b. ♠

```
1 POWER(x, n)
2 if n = 0 then return 1
3 y ← POWER(x, ⌊n/2⌋)
4 z ← y × y
5 if n mod 2 = 0 then return z
6 else return z × x
```

FIG. 9: Calcul de x^n.